SMTCoq: automatisation expressive et extensible dans Coq

Valentin Blot[1,2,3] *†, Amina Bousalem[1], Quentin Garchery[1,2,4], et Chantal Keller[1]

[1] LRI, Univ. Paris-Sud, CNRS UMR8623, Orsay, Université Paris-Saclay
Prénom.Nom@lri.fr
[2] IRIF, Univ. Paris-Diderot, CNRS UMR8243, Paris
[3] LSV, ENS Paris-Saclay, CNRS UMR8643, Cachan, Université Paris-Saclay
[4] Inria, Palaiseau, Université Paris-Saclay

Résumé

Les assistants de preuve basés sur la Théorie des Types, tels que Coq, permettent l'implantation de tactiques automatiques efficaces reposant sur la réflexion calculatoire (ex : `lia`, `ring`). Malheureusement, celles-ci sont généralement limitées à un domaine mathématique particulier (ex : l'arithmétique linéaire entière, la théorie des anneaux). *A contrario*, SMTCoq est un outil modulaire et extensible, faisant appel à des prouveurs externes, qui généralise ces approches calculatoires pour combiner les raisonnements issus de multiples domaines. Pour cela, il repose sur une interface de haut niveau, qui offre une plus grande expressivité, au prix d'une automatisation plus complexe.

Dans cet article, nous détaillons deux améliorations : la possibilité de faire appel à des lemmes quantifiés, et celle d'utiliser plusieurs représentations d'une même structure de données. Elles permettent de construire une tactique automatique basée sur SMTCoq qui soit expressive sans remettre en cause la modularité ni l'efficacité de ce dernier. Une telle tactique permet ainsi une automatisation extensible, à faible coût, à de nouveaux domaines supportés par les prouveurs automatiques de l'état de l'art.

1 Introduction

Les méthodes formelles regroupent un ensemble de techniques et outils permettant de concevoir, développer et vérifier des systèmes informatiques avec un haut niveau de confiance. L'intérêt des méthodes formelles est mis en avant dans le cas des systèmes critiques. Plus généralement, un certain nombre de méthodes nécessitent des outils capables de formuler et de prouver des propriétés mathématiques de manière formelle. Parmi ces outils, on distingue deux grandes familles : les assistants de preuve et les prouveurs automatiques. Les premiers sont expressifs mais nécessitent une intervention importante de l'utilisateur lors de la contruction des preuves. Les prouveurs automatiques, quant à eux, définissent des heuristiques de recherche de preuve ne nécessitant aucune intervention de l'utilisateur, au prix d'une expressivité plus restreinte. SMTCoq (disponible à l'adresse `https://smtcoq.github.io`) fournit un pont entre ces deux familles d'outils en permettant l'appel, au moyen de tactiques dédiées, de prouveurs automatiques au sein de l'assistant de preuve Coq. Pour cela, afin de conserver la cohérence de Coq, les prouveurs automatiques utilisés doivent renvoyer un fichier de certificat justifiant la preuve d'un problème donné.

Initialement, le développement de SMTCoq visait à permettre la vérification de tels certificats SAT ou SMT importants [1] : un programme écrit en Coq, le vérificateur, rejoue les étapes du certificat et vérifie la correction de chacune d'elles, afin de déterminer la validité du problème

*This research was supported by the Labex DigiCosme (project ANR11LABEX0045DIGICOSME) operated by ANR as part of the program "Investissements d'Avenir" Idex ParisSaclay (ANR11IDEX000302).

† This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 799557.

d'entrée. Ce programme est certifié : un lemme de correction énonce que si le vérificateur termine avec succès, alors le problème est prouvé. Ainsi, la confiance dans les résultats des prouveurs automatiques est accrue par la vérification supplémentaire effectuée en Coq.

Les développements récents de SMTCoq [10, 11] se sont attachés à améliorer son expressivité et sa facilité d'utilisation. En effet, SMTCoq possède une interface de haut niveau reposant sur une représentation modulaire des certificats. Ainsi il peut facilement être étendu à de nouveaux prouveurs ainsi qu'à de nouvelles théories supportées par ces prouveurs. SMTCoq a ainsi été étendu au prouveur CVC4 [2] et aux théories des vecteurs de bits et des tableaux supportées par celui-ci. Cela requiert l'extension du format de certificats de SMTCoq, la définition du vérificateur sur ces nouvelles règles et la preuve des lemmes de correction correspondant. Cette preuve requiert la définition des théories supportées par le prouveur au sein de Coq, ou l'utilisation de théories déjà existantes. L'expressivité de SMTCoq peut également être améliorée en permettant d'effectuer des conversions entre les diverses implantions en Coq d'une théorie donnée et la théorie dans laquelle a été implanté le lemme de correction. Pour la théorie des formules booléennes, le lemme de correction utilise le type `bool` des booléens. Le développement d'une tactique de conversion entre ce type et le type `Prop` des propositions de Coq [11] a ainsi permis d'utiliser la machinerie de SMTCoq pour prouver des théorèmes formulés dans le type `Prop`.

Dans cet article, nous exploitons ces techniques dans un cadre général pour étendre encore l'expressivité de SMTCoq. Nos contributions sont les suivantes. D'une part, nous montrons comment étendre le vérificateur à une théorie d'une autre nature : un fragment de la logique du premier ordre. Pour cela nous étendons le format de certificats de SMTCoq, le vérificateur, et le lemme de correction, pour permettre l'instanciation de lemmes universellement quantifiés. Nous appliquons cette extension aux certificats générés par le prouveur veriT. D'autre part, nous définissons de nouvelles tactiques de conversion entre différents types d'entiers dans Coq. Ainsi, bien que la théorie arithmétique entière de SMTCoq utilise le type de données `Z` de Coq, il est maintenant également possible d'utiliser des représentations alternatives telles que `N`, `nat` ou `positive`. Ces améliorations font bénéficier les utilisateurs de Coq d'une meilleure automatisation au sein de l'assistant de preuve tout en rendant l'outil plus facile à utiliser.

Cette automatisation peut être appliquée à de nombreux cas du premier ordre. Par exemple, en axiomatisant la théorie des groupes avec un élément neutre à gauche `e` et un inverse à gauche, la tactique `verit` fournie par SMTCoq (faisant appel au prouveur SMT veriT) permet de prouver automatiquement que `e` est neutre à droite, et que l'inverse est un inverse à droite. En axiomatisant également l'itéré de la multiplication `power`, on peut également simplement prouver

```
Goal forall (n:nat), power e n = e.
```

en réalisant une induction sur `n`, puis en appliquant la tactique `verit`.

Dans la section suivante nous donnons un aperçu du fonctionnement général de SMTCoq ainsi que les éléments nécessaires à la présentation de notre travail. Dans la section 3 nous présentons l'extension de SMTCoq pour la gestion de lemmes quantifiés puis, dans la section 4, nous présentons les tactiques de conversion entre représentations d'entiers en Coq. Nous finissons par une discussion sur les travaux connexes avant de conclure et donner nos perspectives de recherche futures.

2 SMTCoq et son environnement

Afin de mieux comprendre comment SMTCoq établit une communication entre l'assistant de preuve Coq et différents prouveurs automatiques, nous commençons par détailler le fonctionnement de ces logiciels formels. On s'intéresse ensuite plus précisément à SMTCoq en donnant sa décomposition en modules de programmation.

2.1 L'assistant de preuve Coq

Les assistants de preuve permettent d'exprimer des théorèmes complexes puis de les vérifier de manière interactive. Un assistant de preuve est en ce sens un outil permettant de vérifier les étapes d'une preuve fournie de manière rigoureuse et exhaustive par l'utilisateur.

L'assistant de preuve Coq s'appuie sur l'isomorphisme de Curry-Howard et implante la vérification de preuves au moyen d'un algorithme de typage pour le calcul des constructions inductives [16]. Celui-ci constitue le *noyau* de Coq, le composant critique sur lequel repose la correction de l'assistant de preuve dans son ensemble. Afin de maximiser la confiance des utilisateurs dans ce noyau, celui-ci est implanté en OCaml, un langage de haut niveau proche de sa logique. L'accent est mis sur la concision et la clarté du code.

Les structures de données en Coq sont définies au moyen de types inductifs. Le type simplifié des formules de SMTCoq est donné par :

```
Inductive formula :=
    | Bool (b : bool)
    | Neg (f : formula)
    | And (f1 : formula) (f2 : formula)
    | ...
.
```

Un élément du type `formula` représente un arbre avec des booléens aux feuilles, les nœuds étant des connecteurs logiques. Dans SMTCoq, en plus des booléens, le type des formules inclut les symboles d'arithmétique linéaire sur les entiers, les symboles de la théorie des tableaux et de la théorie des vecteurs de bits mais ne contient pas de quantificateurs.

L'interprétation est une fonction qui calcule la valeur d'une formule de SMTCoq et est définie par induction :

```
Fixpoint interp t := match t with
    | Bool b => b
    | Neg f => negb (interp f)
    | And f1 f2 => andb (interp f1) (interp f2)
    | ...
end.
```

Les fonctions Coq `negb` et `andb` implantent respectivement la négation booléenne et la conjonction booléenne. Enfin, il est possible de prouver des propriétés reposant sur ces définitions :

```
Lemma andproj1 f1 f2 :
  interp (And f1 f2) = true -> interp f1 = true.
```

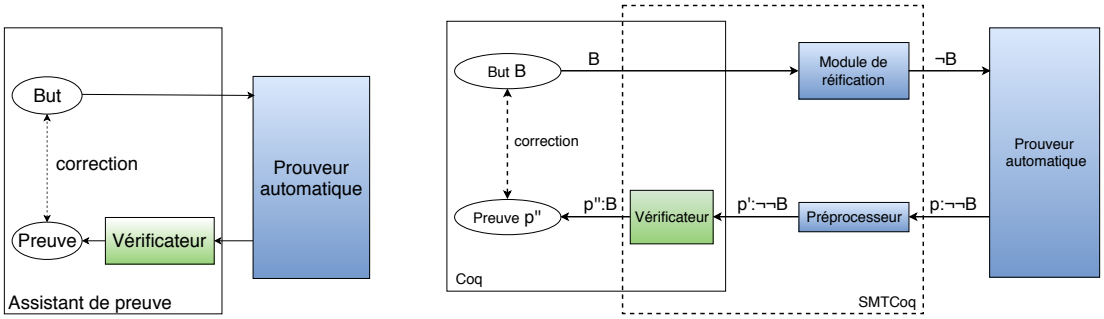


FIGURE 1 – Approche sceptique : cas général (gauche) et application à SMTCoq (droite)

2.2 Les prouveurs automatiques

Les prouveurs automatiques définissent des heuristiques de recherche de preuve et ne reposent donc pas sur l'utilisateur pour la construction de la preuve : l'effort de certification se réduit à la formalisation du problème. En contrepartie, la logique d'un prouveur automatique est plus limitée et/ou la réponse en temps fini n'est pas garantie. L'efficacité de la recherche de preuve étant primordiale, les prouveurs automatiques sont fréquemment écrits dans des langages bas niveau tels que C ou C++, font usage de structures mutables complexes et contiennent généralement de nombreuses optimisations.

Parmi les prouveurs automatiques, les prouveurs SAT (*satisfiabilité* propositionnelle) et SMT (*satisfiabilité modulo théories*) prennent en entrée des formules contenant des variables et tentent de résoudre le problème de satisfiabilité de ces formules : il s'agit de savoir s'il existe une affectation des variables rendant toutes les formules vraies. Dans le cas où une telle affectation existe, la réponse consiste en l'affectation en question. Dans le cas contraire, certains prouveurs fournissent une preuve qu'aucune affectation des variables ne rend les formules vraies. Cette preuve se présente sous la forme d'un fichier de certificat dont le format peut varier en fonction du prouveur automatique considéré.

2.3 Approche sceptique

L'utilisation de prouveurs automatiques au sein d'un assistant de preuve peut se faire selon deux méthodes appelées *sceptique* et *autarcique*. Alors que l'approche autarcique repose sur une certification de l'ensemble du code du prouveur automatique, l'approche sceptique se contente d'obtenir et vérifier un certificat de la part du prouveur automatique.

L'approche sceptique, utilisée par SMTCoq, nécessite seulement la certification d'un vérificateur de certificats plutôt que de l'ensemble du prouveur automatique, comme représenté sur la FIGURE 1 (gauche). À chaque appel du prouveur automatique, le certificat éventuellement généré par celui-ci est vérifié puis transformé en une preuve du but initial. Cette approche ne permet pas de garantir la complétude du système : certains buts valides ne sont pas démontrés, notamment si le prouveur automatique renvoie un certificat erroné ou ne renvoie pas de certificat. En revanche, le développement du prouveur automatique reste indépendant de son utilisation dans l'assistant de preuve puisque seuls les certificats qu'il produit sont vérifiés.

2.4 Fonctionnement de SMTCoq

L'utilisation de l'approche sceptique permet à SMTCoq d'être modulaire vis-à-vis des prouveurs automatiques : SMTCoq supporte aujourd'hui les prouveurs zChaff, veriT et CVC4 et peut être étendu à d'autres prouveurs simplement en écrivant l'analyseur syntaxique pour le format des certificats fournis par celui-ci. Ainsi, la version actuelle de SMTCoq fournit les tactiques `zchaff`, `verit` et `cvc4` qui permettent de faire appel au prouveur correspondant pour résoudre le but courant et ainsi profiter de l'automatisation du prouveur au sein de l'assistant de preuve.

Décrivons maintenant les différentes étapes effectuées par SMTCoq lorsque l'utilisateur lance l'une des tactiques fournies et que le but courant est, par exemple, $\forall x\, B(x)$ (où B est une formule sans quantificateur). SMTCoq envoie tout d'abord la négation de la proposition B au prouveur automatique. Si ce dernier renvoie un certificat indiquant la non-satisfiabilité du problème, on en déduit une preuve du lemme. En effet, puisque les prouveurs automatiques résolvent des problèmes de satisfiabilité (quantificateurs existentiels), pour prouver un lemme en forme prénexe (quantificateurs universels) tel que $\forall x\, B(x)$, on se ramène à $\neg(\exists x\, \neg B(x))$. Ces deux formules sont équivalentes dans notre cas car nous requérons que tous les prédicats apparaissant dans B soient décidables [1].

La construction d'une preuve du but initial par SMTCoq passe par plusieurs étapes (Figure 1 droite). L'énoncé du lemme est d'abord traduit par le module de réification (§ 2.6) dans un type de données OCaml. Le terme obtenu peut alors être écrit dans un fichier servant d'entrée au prouveur automatique. En cas de succès de ce prouveur, on obtient un fichier de certificat qui est ensuite traduit par le préprocesseur (§ 2.7) dans un format adapté au vérificateur de SMTCoq. Ce dernier rejoue la preuve en Coq (§ 2.5).

2.5 Vérificateur certifié

Le vérificateur est une fonction `checker` (§ 2.5.1) écrite et certifiée en Coq qui reproduit le fonctionnement des certificats des prouveurs automatiques. La preuve du but initial repose sur le théorème de correction de cette fonction (§ 2.5.2).

2.5.1 La fonction `checker`

La fonction `checker` a pour type `formula -> list rule -> bool` où le premier argument de cette fonction est la formule d'entrée du problème. Le deuxième argument est un certificat formalisé en Coq, défini comme une liste de règles. La valeur de retour de `checker` indique s'il est possible d'obtenir une contradiction à partir de la formule et du certificat comme précisé dans le paragraphe suivant. Le certificat Coq est obtenu à partir d'un fichier de certificat fourni par un prouveur automatique, chaque règle de ce dernier étant traduite en une règle Coq, dont le type est donné par :

```
Inductive rule :=
    | AndProj (pos_prem : int) (ind_proj : int)
    | Resolution (pos_param : int list)
    | ...
.
```

Afin de calculer sa valeur de retour, la fonction `checker` maintient un ensemble de formules appelé état (que nous représentons ici, pour plus de clarté, à l'aide d'une liste) qui, initialement,

1. Nous n'ajoutons donc pas d'axiome classique en Coq, $\neg(\exists x\, \neg B(x)) \Rightarrow \forall x\, B(x)$ étant alors prouvable.

contient la formule d'entrée. Prenons pour exemple le problème ayant pour formule d'entrée `in` définie par

```
in := And (Bool x) (Neg (Bool x))
```

où `x` est un booléen. L'état est alors initialisé à `[in]`. Pour chaque règle du certificat Coq, la fonction `checker` enrichit l'état d'une nouvelle formule définie par la règle en question.

En appelant un prouveur automatique sur l'exemple, on obtient un fichier de certificat que l'on peut traduire en un certificat Coq :

```
c := [AndProj 1 1; AndProj 1 2; Resolution [3; 2]]
```

Dans le cas de la règle `AndProj`, le paramètre `pos_prem` indique la position dans l'état de la prémisse. La règle est correctement appliquée si cette formule est de la forme `And t1 t2`. Dans ce cas, l'indice de projection `ind_proj` est utilisé pour savoir quelle projection appliquer pour obtenir la formule à rajouter à l'état. La première règle du certificat donné en exemple a pour prémisse la formule en première position de l'état, c'est-à-dire `And (Bool x) (Neg (Bool x))`. En faisant la projection sur la première composante de la conjonction, on sait que `checker` ajoute la formule `Bool x` à l'état. Après cette règle, l'état est donc `[in; Bool x]`. De même, la deuxième règle ajoute `Neg (Bool x)` à l'état.

La règle de résolution prend une liste de paramètres en argument qui représente les emplacements de l'état à prendre en compte. Retenons simplement que si les formules présentes à ces positions contiennent une formule et sa négation alors `checker` ajoute `Bool false` à l'état. Ainsi, après la dernière règle du certificat, l'état devient `[in; Bool x; Neg (Bool x); Bool false]`.

Enfin, la valeur de retour de `checker` est un booléen qui indique si, après avoir pris en compte le certificat, l'état contient la formule `Bool false`. C'est le cas de notre exemple, on a donc `checker in c = true`.

2.5.2 Le théorème de correction

Pour obtenir une preuve du but initial, on applique le théorème de correction qui, pour une formule `in` et un certificat `c`, nous donne :

```
checker in c = true -> interp in = false
```

On notera que ce théorème donne bien la négation de la formule d'entrée (§ 2.4).

La preuve de ce théorème repose sur le *lemme de correction d'une règle* qui nous assure que `checker` ajoute une nouvelle formule valide dans l'état courant (on dit qu'une formule est valide lorsque son interprétation est vraie). Ainsi, lorsque l'on veut modifier une règle ou rajouter un nouveau type de règle, il suffit de modifier ou de compléter la preuve de ce lemme. Ce fonctionnement facilite le développement incrémental de SMTCoq, qui va nous permettre d'ajouter le support pour des lemmes quantifiés (section 3).

Donnons les grandes étapes de la preuve du théorème de correction. Étant donnés un certificat `c` et une formule `in`, on suppose à la fois que `checker in c = true` et que `in` est valide et on veut aboutir à une contradiction. Puisque l'état est initialisé avec la formule d'entrée `in`, toutes les formules de l'état sont valides. On sait, grâce au lemme de correction d'une règle, que cette propriété est conservée par `checker`. Comme on a par ailleurs `checker in c = true`, c'est-à-dire que l'état final contient la formule `Bool false`, l'interprétation de cette formule fournit la contradiction voulue.

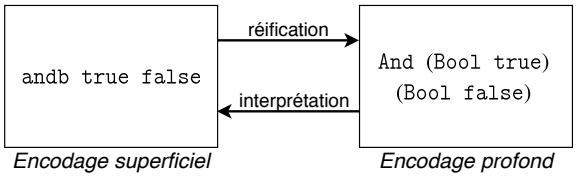
La preuve de l'hypothèse `checker in c = true` est obtenue par calcul de la fonction `checker`. La preuve du but initial repose donc sur un calcul, c'est ce qu'on appelle la réflexion calculatoire.

2.6 Réification

La réification est le fait de passer d'un encodage superficiel (*shallow-embedding*) à un encodage profond (*deep-embedding*). Dans le cas d'un encodage profond, un terme est un élément d'un type de données (comme le type `formula`) ce qui met en évidence sa structure. À l'inverse, l'encodage superficiel du même terme est donné directement par sa valeur dans le langage (ici, Coq).

Donnons l'exemple de la formule $true \ \& \ false$ où & désigne la conjonction booléenne. On peut donner son encodage superficiel à l'aide de la fonction Coq `andb` qui implante la conjonction booléenne : `andb true false`. L'encodage profond de la formule peut être donné dans le type inductif `formula` : `And (Bool true) (Bool false)`.

Pour pouvoir utiliser la fonction de réification, il faut que son résultat soit lié au terme initial : c'est le rôle de la fonction d'interprétation (§ 2.1). Cette dernière doit inverser la réification de sorte que, pour un terme Coq `t`, on ait `t = interp u` où `u` est la réification de `t`.



La réification, dans le cas de SMTCoq, sert à mettre en évidence la structure des formules afin de pouvoir les écrire dans un fichier au format reconnu par un des prouveurs automatiques disponibles et de permettre à la fonction `checker` de les manipuler.

2.7 Préprocesseur

Les certificats passent par une étape intermédiaire assurée par le préprocesseur avant d'être interprétés en des certificats SMTCoq. Cette étape présente plusieurs avantages.

Premièrement, le préprocesseur permet d'assurer la modularité de SMTCoq vis-à-vis des prouveurs automatiques. Bien que les certificats puissent avoir différentes formes en fonction du prouveur automatique dont ils proviennent, il sont tous traduits dans le format commun du paragraphe 2.5.1. Actuellement, le préprocesseur prend en charge trois prouveurs automatiques, à savoir : zChaff, veriT et CVC4.

Une fois cette traduction dans un format commun faite, le préprocesseur optimise le certificat obtenu. Par exemple, afin d'avoir un certificat de petite taille, le préprocesseur élimine les règles redondantes. Il calcule également l'allocation des formules dans l'état du vérificateur (un emplacement contenant une formule qui n'est plus utile peut être réutilisé).

Enfin, il n'est pas nécessaire de prouver la correction du préprocesseur. En effet, on utilise l'approche sceptique qui consiste à vérifier que le certificat fourni est correct et non à verifier que le procédé de génération du certificat est correct.

3 Ajout de lemmes quantifiés

3.1 Motivations

Initialement, SMTCoq permettait uniquement la démonstration automatique de théorèmes en forme prénexe : il n'était pas possible de faire appel à un théorème quantifié déjà démontré.

Par exemple, en supposant que l'on a réussi à montrer les deux lemmes suivants :

$$\forall h.homme(h) \Rightarrow mortel(h) \qquad\qquad homme(Socrate)$$

alors on ne peut pas montrer automatiquement (en utilisant SMTCoq) que :

$$mortel(Socrate)$$

Cette partie détaille une amélioration de la modularité de SMTCoq qui consiste à permettre la transmission de lemmes (potentiellement) quantifiés au prouveur automatique.

3.2 Ajout de lemmes au contexte

L'ajout de lemmes quantifiés au contexte demande d'étendre les étapes de construction d'une preuve Coq à partir du but initial (Figure 2).

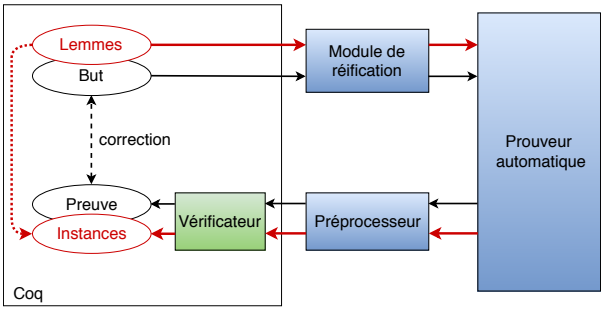


Figure 2 – Ajout de lemmes

En plus du but initial, le module de réification envoie au prouveur automatique les lemmes que l'utilisateur a rajoutés au contexte. Le préprocesseur traduit la réponse du prouveur en un certificat dont le format aura été étendu pour prendre en compte les instanciations de lemmes (§ 3.4.2). Le vérificateur en déduit une preuve qui dépend d'instances de ces lemmes (§ 3.3). Enfin, il faut trouver des preuves de ces instances à partir des lemmes (§ 3.4.3).

Cet ajout a l'avantage de ne pas étendre les formules de SMTCoq avec des quantificateurs, et ne demande donc que de légères modifications au niveau du vérificateur.

3.3 Vérificateur pour l'ajout de lemmes quantifiés

On se propose ici de compléter le type `rule` en laissant inchangé le type des formules. Cette modification vise à prendre en charge les lemmes en forme prénexe ajoutés au contexte par l'utilisateur. Le cas général où les quantificateurs peuvent être dans n'importe quelle position dans la formule ne peut pas être traité avec cette approche : il faudrait étendre le type `formula` avec des quantificateurs, ce qui nous obligerait à étendre la fonction d'interprétation et à adapter les preuves la concernant.

Le nouveau constructeur du type `rule` s'écrit :

```
ForallInst (lemma : Prop) (plemma : lemma)
           (inst : formula) (pinst : lemma -> interp inst = true)
```

Lorsque `checker` rencontre cette règle, l'instance `inst` est ajoutée à l'état. Les autres champs de cette règle sont nécessaires pour la preuve du lemme de correction d'une règle. Ainsi `lemma` devra être un lemme en rapport avec cette instance :

— ce lemme doit être prouvé, la preuve est le champ `plemma`

— ce lemme doit nous assurer que l'instance est valide, c'est le champ `pinst`

En effet, on peut prouver que `inst` est valide, et même indépendamment des formules contenues dans l'état. Il suffit pour cela d'appliquer `pinst` à `plemma`. On a donc rétabli la preuve du théorème de correction.

Cette règle permet de traiter les instanciations des prouveurs automatiques mais demande, pour une instance donnée, d'identifier un lemme, sa preuve et de donner une preuve d'instanciation (champ `pinst`). Cet aspect de la construction d'une règle `ForallInst` est spécifique au prouveur automatique considéré, le cas d'étude présenté résout le cas de veriT (§ 3.4.3). La preuve d'instanciation est donnée à l'aide d'une coupure (tactique `assert` en Coq), ce qui nous permet de l'expliciter dans un deuxième temps.

La règle `ForallInst` diffère des règles utilisées dans SMTCoq jusqu'ici : elle utilise l'encodage superficiel pour le lemme Coq et l'encodage profond pour l'instance. D'un côté, l'utilisation de lemmes Coq déjà existants ne nécessite pas de les réifier, ce qui justifie l'utilisation d'un encodage superficiel. D'un autre côté, l'encodage profond des formules est déjà implanté pour les termes en provenance du fichier de certificat.

3.4 Cas d'étude : application au prouveur veriT

Le format commun des termes de SMTCoq permet d'encoder les certificats des différents prouveurs automatiques sous un même type. L'extension du vérificateur qui est proposée ici est donc utilisable pour n'importe quel prouveur automatique à condition d'étendre le préprocesseur pour celui-ci.

Dans cette partie, nous allons voir comment la règle d'instanciation $forall\_inst$ de veriT est encodée dans le format commun. Cet encodage nécessite de transformer le certificat veriT, de retrouver à quel lemme correspond une instance et de montrer que ce lemme implique l'instance.

3.4.1 Instanciation d'un lemme par veriT

Les certificats de veriT sont constitués d'une suite de règles de la forme :

$$id : (typ\ res\ prem)$$

où id est un entier qui identifie la règle, typ est le type de la règle, la formule res est la conclusion de la règle et $prem$ est la liste des identifiants des prémisses de la règle.

Prenons en exemple le fichier de certificat suivant :

$1 : (input\ (\neg((3 + a) * b = 3 * b + a * b))$

$2 : (input\ (\forall x\, y\, z,\ (x + y) * z = x * z + y * z))$

$3 : (forall\_inst\ (\neg(\forall x\, y\, z,\ (x + y) * z = x * z + y * z) \vee (3 + a) * b = 3 * b + a * b))$

$4 : (resolution\ ()\ 3\ 2\ 1)$

Les règles *input* correspondent aux formules données en entrée. La règle 4 est une règle de résolution qui a pour prémisses les conclusions des règles 1, 2 et 3 et qui en déduit l'absurde représenté par (). La règle 3 est la règle utilisée par veriT pour instancier un lemme. On remarquera que cette règle n'a pas de prémisse et que l'utilisation de son résultat se fait au moyen de la règle de résolution.

3.4.2 Préprocesseur pour la règle *forall_inst*

Le résultat de la règle *forall_inst* est de la forme $\neg(\forall x, P\ x) \vee (P\ c)$ alors que lorsque `checker` rencontre la règle `ForallInst` c'est seulement l'instance `P c` qui est enregistrée dans l'état. Puisqu'on veut encoder la règle *forall_inst* de veriT par la règle Coq `ForallInst`, il faut modifier la suite du certificat qui dépend de cette règle. La forme de la conclusion de la règle *forall_inst* indique qu'elle ne sera utilisée dans la suite du certificat que dans une ou plusieurs règles de résolution (ce qui se vérifie en pratique). On se ramène donc à modifier seulement les règles de résolution.

Par exemple, le fichier certificat de l'exemple donné dans le paragraphe 3.4.1 est traduit dans un certificat Coq de la forme :

```
[ForallInst _ _ inst _; Resolution 4 [1; 3]]
```

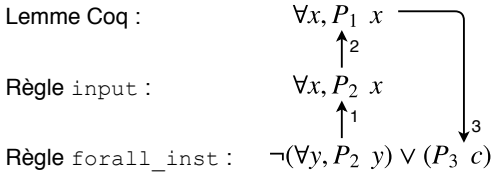
où les _ sont à compléter (voir paragraphe suivant) et où l'instance `inst` est donnée par :

```
Eq (Mult (Plus 3 a) b) (Plus (Mult 3 b) (Mult a b))
```

On remarque que la règle `Resolution` n'a plus de dépendance à la formule $\forall x\, y\, z,\ (x+y)*z = x*z+y*z$ donnée en entrée.

3.4.3 Construction d'une règle `ForallInst`

La construction d'une règle `ForallInst` demande de reconnaître à quel lemme correspond une instance (champs `lemma` et `plemma`) et de montrer que ce lemme implique bien l'instance (champ `pinst`).



Reconnaissance du lemme correspondant à l'instance. La reconnaissance du lemme passe par une étape intermédiaire : on commence par chercher de quelle règle *input* vient l'instanciation (flèche `1` dans la figure). L'avantage de cette étape est que la formule dans la règle *input* est plus proche du lemme Coq, en particulier les variables liées n'ont pas été renommées.

On cherche ensuite à établir un lien entre une formule du certificat de veriT et un lemme rajouté par l'utilisateur de SMTCoq (flèche `2` dans la figure). La difficulté vient du fait que les lemmes additionnels peuvent apparaître modifiés dans les certificats de veriT. Par exemple, si un lemme Coq a une sous-formule de la forme $a = b$, alors le résultat de la règle *input* correspondant à ce lemme peut avoir à la place $b = a$ comme sous-formule. Rétablir la forme initiale est une solution trop coûteuse car il faudrait modifier la structure de toutes les formules suivantes qui en dépendent. Pour résoudre ce problème, on utilise une table de hachage, initialisée avec toutes les sous-formules des lemmes Coq, qui nous sert pour reconnaître des formules aux modifications de veriT près. Pour traiter l'exemple de la symétrie de l'égalité, à chaque fois que l'on rencontre une sous-formule de la forme $a = b$, en plus de vérifier si cette formule est déjà contenue dans la table, on regarde aussi si la formule $b = a$ est dans cette même table. Ainsi, on peut reconnaître

efficacement des formules identiques modulo symétrie de l'égalité et retrouver à quel lemme se rapporte une formule.

La recherche du lemme correspondant à l'instance à été implantée en OCaml et nous permet de compléter les champs `lemma` et `plemma` lors de la construction d'une règle `ForallInst`.

Preuve automatique d'une instanciation. Pour compléter la construction de la règle `ForallInst`, il faut donner la preuve d'instanciation (flèche `3` dans la figure ci-dessus). On définit donc une tactique Coq qui permet d'automatiser la recherche de ces preuves.

Le lemme Coq suivant correspond directement à l'instanciation du lemme donné :

$$(\forall x, P\ x) \Rightarrow P\ c$$

Dans ce cas simple, la tactique `auto` résout automatiquement le but. Cependant, ce n'est plus le cas lorsque l'instance est légèrement modifiée. Par exemple, le lemme suivant ne peut pas être prouvé directement par la tactique `auto` :

$$(\forall x, f\ x = f\ c) \Rightarrow f\ c = f\ 3$$

Il faut donc remettre le but dans une forme qui correspond à celle du lemme. Dans le cas ci-dessus il faut utiliser la symétrie de l'égalité avant d'appliquer la tactique `auto`.

Deux sources de modifications sont à prendre en compte ici (voir figure). D'une part le lemme en entrée dans le fichier de certificat peut différer du lemme Coq initial (on peut avoir $P_1 \neq P_2$ comme expliqué précédemment). D'autre part, certaines modifications de veriT sont propres à la règle d'instanciation (on peut aussi avoir $P_2 \neq P_3$).

La solution proposée se présente sous la forme d'une tactique Coq et prend en compte toutes ces modifications afin de trouver automatiquement les preuves d'instanciation.

3.5 Résultats et perspectives

Nous avons présenté une extension de la tactique `verit` à l'utilisation de lemmes déjà démontrés. Ceux-ci sont ajoutés au contexte au moyen d'une instruction dédiée `Add_lemmas` ou passés en paramètre à la tactique. Lorsqu'aucun lemme n'est ajouté, son comportement reste le même que précédemment. Dans le cas où des lemmes sont ajoutés au contexte et que le prouveur automatique instancie ces lemmes, les instanciations sont automatiquement prouvées comme décrit ci-dessus.

L'ajout de lemmes s'est traduit par une extension de la logique utilisée et se retrouve dans toutes les étapes intermédiaires de SMTCoq, étapes qu'il a fallu étendre en conséquence. Cette extension a nécessité un développement original : la technique d'encodage des instanciations des lemmes utilisée simplifie l'extension du vérificateur (§ 3.3). Enfin, le procédé de vérification final a été automatisé (§ 3.4.3) afin de préserver la facilité d'utilisation de SMTCoq. Grâce à la modularité de SMTCoq, appliquer l'approche à d'autres prouveurs (capable d'instancier des lemmes et de fournir les instances) consiste simplement à compléter le pré-processeur comme cela a été fait pour `verit`, sans avoir besoin de modifier le code Coq.

Pour valider notre approche, nous avons automatiquement prouvé des propriétés plus complexes grâce à SMTCoq, portant sur : la théorie des groupes, une théorie formalisant les listes d'entiers, des fonctions définies récursivement, etc.

Une extension serait de pouvoir ajouter au contexte des lemmes qui ne sont pas nécessairement en forme prénexe. Cette extension de la logique de SMTCoq ne peut pas être réalisée directement avec la méthode proposée ici, qui requiert que les instances des lemmes soient sans quantificateurs, et est donc laissée pour une perspective à plus long terme.

Afin que l'utilisateur n'ait pas à fournir les lemmes à utiliser, tout en évitant de surcharger le prouveur avec tous les lemmes présents dans le contexte, on pourrait aussi utiliser des méthodes de *machine learning* pour sélectionner automatiquement un sous-ensemble des lemmes, comme c'est fait dans CoqHammer et SledgeHammer [4, 9].

4 Traductions entre représentations des données

4.1 Motivations

En Coq, comme dans la plupart des assistants de preuve, plusieurs représentations cohabitent pour une même structure de données. Le problème est bien connu des utilisateurs de Coq pour la représentation des entiers : les bibliothèques usuelles définissent notamment les entiers naturels unaires (type `nat`) et binaires (type `N`), les entiers strictement positifs binaires (type `positive`), les entiers relatifs binaires (type `Z`), les grands entiers en base 2^{31} (types `bigN` et `bigZ`), ...etc. Ce problème se pose pour la plupart des types de données : par exemple, les vecteurs de bits peuvent être représentés avec des types dépendants de la longueur ou non.

Pour utiliser la pleine potentialité des prouveurs sous-jacents, SMTCoq doit injecter toutes les représentations d'un même type de données dans le type SMT correspondant. Plusieurs approches ont été proposées pour effectuer automatiquement de telles conversions (voir la section 5 pour une présentation détaillée), mais un problème nouveau se pose dans le cas de SMT : la conversion doit pouvoir être effectuée dans des termes combinant plusieurs théories, et non uniquement la théorie correspondant à la structure de données considérée.

Par exemple, le but suivant, mêlant la théorie de l'arithmétique (sur les entiers strictement positifs) avec un symbole de fonction non interprété `f` :

```
x, y : positive                          f : positive -> positive
============================================================
((x + 3) =? y) -> ((3 <? y) && ((f (x + 3)) <=? (f y)))
```

doit être converti [2] en un but similaire sur les entiers relatifs :

```
x', y' : Z          Hx' : 0 <? x'       Hf'x' : 0 <? f' (x' + 3)
f' : Z -> Z         Hy' : 0 <? y'       Hf'y' : 0 <? f' y'
============================================================
((x' + 3) =? y') -> ((3 <? y') && (f' (x' + 3) <=? f' y'))))
```

où, cette fois, les symboles `+`, `=?`, `<=?` et `<?` et la constante `3` sont ceux du type `Z`, type que la plupart des prouveurs SMT ainsi que SMTCoq supportent déjà [1].

Dans cette partie, nous détaillons une nouvelle approche tenant compte de cette particularité, et là encore peu intrusive vis-à-vis de SMTCoq. Pour des raisons de clarté, nous la présentons ici sur le cas de l'injection du type `positive` dans le type `Z`, mais l'implantation est générique pour toute injection dans `Z` (réalisée à l'aide d'un foncteur, qui est instancié pour les types `nat`, `N` et `positive`) et peut se généraliser similairement à d'autres structures de données.

4.2 Principe

L'approche retenue consiste à transformer le but avant tout appel aux prouveurs automatiques. Cette transformation est effectuée par une tactique écrite en Ltac, le langage de tactiques

2. Plus généralement, on pourrait souhaiter l'ajout de l'hypothèse `forall z, 0 <? z -> 0 <? f' z`. Nous laissons cela en perspective.

de Coq, ce qui permet de profiter du moteur de réécriture de Coq. Dans l'architecture de SMTCoq (FIGURE 1 droite), cela se positionne donc avant le module de réification. La transformation se déroule en trois étapes :

1. l'application d'une double conversion à un sous-ensemble des objets de type `positive` ;
2. l'application de réécritures permettant de convertir les symboles de la théorie des entiers du type `positive` vers le type `Z` ;
3. le renommage des sous-termes résiduels, avec l'ajout des propriétés liées au fait que l'injection de `positive` dans `Z` n'est pas surjective.

Nous allons détailler le fonctionnement sur l'exemple du but indiqué ci-dessus.

1. Double conversion La première étape applique récursivement une double conversion à un sous-ensemble des sous-termes du but de type `positive`, en vue d'obtenir le but suivant :

```
(Z2Pos (Pos2Z x) + Z2Pos (Pos2Z 3) =? Z2Pos (Pos2Z y)) ->
  ((Z2Pos (Pos2Z 3) <? Z2Pos (Pos2Z y)) &&
   (Z2Pos (Pos2Z (f (Z2Pos (Pos2Z x) + Z2Pos (Pos2Z 3))))
                       <=? Z2Pos (Pos2Z (f (Z2Pos (Pos2Z y)))))).
```

L'injection `Pos2Z : positive -> Z` injecte simplement les entiers strictement positifs dans les entiers relatifs. Son inverse partiel `Z2Pos : Z -> positive` injecte les entiers relatifs strictement positifs dans le type `positive`, et renvoie une valeur quelconque dans les autres cas. En effet, ces cas ne sont pas utilisés car la propriété essentielle est que la double conversion de type `positive -> positive` soit l'identité :

```
Lemma Pos2Z_id : forall (p : positive), p = Z2Pos (Pos2Z p).
```

Afin de pouvoir appliquer cette réécriture, cette étape recherche, de manière répétée, tous les contextes du but contenant un terme de type `positive`, et applique la double conversion à chaque sous-terme voulu :

```
repeat
  match goal with
  | |- appcontext C[?p]  =>
    lazymatch type of p with
    | positive => if forme_voulue C p
                  then rewrite (Pos2Z_id p) else fail
    end
  end
```

où `lazymatch` est une variante de `match` permettant ici plus d'efficacité.

L'originalité de cette tactique réside dans la manière dont la tactique teste si le contexte `C` et le terme `p` sont de la forme voulue (noté abusivement `forme_voulue C p` ci-dessus). En effet, on ne souhaite pas appliquer la double conversion si :

1. le symbole de tête de `p` est un symbole arithmétique (car celui-ci sera transformé lors de la deuxième étape), ou l'injection `Z2Pos` (pour que la procédure termine) ; ou
2. le terme n'est pas déjà dans un contexte de la forme `Z2Pos (Pos2Z _)` (même raison).

Pour le cas 1, il suffit de regarder le symbole de tête de `p`. Pour le cas 2, il faut regarder si le contexte `C[]` est de la forme `C'[Pos2Z (ZPos [])]`. Pour cela, on introduit une variable fraîche `var` qui sert à inspecter le contexte sans interférer avec `p` :

```
lazymatch context C[var] with
| context [Z2Pos (Pos2Z var)] => fail
| _ => idtac
end
```

2. Réécriture et 3. Renommage Une fois le but mis sous cette forme, les deux étapes suivantes sont directes.

La deuxième étape réécrit, de manière répétée, les théorèmes établissant les liens entre les opérateurs et les prédicats arithmétiques sur le type `positive` et leurs contreparties sur le type `Z`, afin d'obtenir le but suivant :

```
((Pos2Z x + 3) =? (Pos2Z y)) -> ((3 <? Pos2Z y) &&
   (Pos2Z (f (Z2Pos (Pos2Z x + 3))) <=? Pos2Z (f (Z2Pos (Pos2Z y)))))
```

Cette étape déplace donc la conversion `Z2Pos` le plus à l'extérieur possible, c'est-à-dire sous les appels de fonctions non interprétées (comme `f` dans l'exemple).

Enfin, la troisième étape renomme les sous-termes résiduels : `Pos2Z x`, `Pos2Z y` et `fun a => Pos2Z (f (Z2Pos a))`, en conservant la propriété que, les variables étant injectées de `positive` dans `Z`, elles sont strictement positives. On obtient ainsi le but voulu.

4.3 Résultats et perspectives

Cette méthode permet de traduire des buts entre deux types de données représentant une même structure ou sous-structure, avec les avantages suivants :

— elle agit en présence de symboles de fonctions ne faisant pas partie de la structure considérée (ceux-ci sont alors convertis en symboles non interprétés) ;
— les types n'ont pas besoin d'être isomorphes : il suffit d'une injection ;
— l'effort de démonstration est assez faible, et porte essentiellement sur les preuves d'équivalence entre les symboles interprétés.

Elle a été appliquée avec succès dans SMTCoq pour traduire la plupart des types d'entiers dans le type `Z`. Une perspective à court terme est de l'appliquer aux autres théories, notamment les vecteurs de bits et les tableaux, pour lesquels plusieurs représentation cohabitent également. Cela nécessite l'abstraction ces théories dans les tactiques de conversion. Une perspective à plus long terme est de s'appuyer sur les mécanismes des classes de type, comme présenté dans [18].

5 Travaux connexes

CoqHammer [9] est un plug-in pour Coq proposant une tactique automatique qui, à partir d'un but, sélectionne un ensemble de lemmes, appelle plusieurs prouveurs externes, et vérifie leurs résultats grâce à un prouveur du premier ordre certifié en Coq. Cette approche a l'avantage d'être robuste et a montré de bons résultats sur la bibliothèque standard de Coq. Cela repose sur une axiomatisation des théories (utilisation de lemmes lors de la vérification) qui est moins efficace que la combinaison de procédures de décision et la réflexion calculatoire proposées par SMTCoq. Par ailleurs, CoqHammer s'applique essentiellement à des buts peu combinatoires, alors que SMTCoq permet également de prouver très efficacement des problèmes demandant un raisonnement combinatoire important [1, 13].

CoqHammer tire ses idées de SledgeHammer [17], un outil similaire pour l'assistant de preuve Isabelle/HOL. La première version de SledgeHammer était basée sur la vérification de traces SMT [5], de manière similaire à SMTCoq.

Une alternative est l'approche autarcique, consistant à prouver la correction d'un prouveur dans son ensemble. Cette approche a été menée avec succès par exemple en Coq avec ergo [14] ou dans les prouveurs de type HOL avec metis [12]. Bien qu'ayant l'avantage de prouver la correction des algorithmes sous-jacents, cette approche fige une implantation qui est ainsi difficilement améliorable. Comme indiqué ci-dessus, de tels prouveurs certifiés sont utilisés au sein de SledgeHammer et CoqHammer.

S. Boulmé et A. Maréchal ont proposé une nouvelle approche, appelée la *certification défensive* [8, 7], pour générer un prouveur correct par construction à partir d'un vérificateur de preuves. Cette approche très prometteuse est en cours d'exploration pour SMTCoq.

Plusieurs tactiques proposent des plongements entre types de données, appliqués notamment aux entiers. Dans Coq, les plus utilisées sont `zify` [15], `ppsimpl` [3] et les *transfer tactics* [18]. À notre connaissance, les tactiques `zify` et `ppsimpl` ne sont pas compatibles avec le fait de combiner des théories ; notamment, elles n'opèrent pas sous les symboles de fonctions non interprétés car venant d'autres théories. Elles ne sont donc pas utilisables dans notre cadre. Les *transfer tactics* pourraient opérer sous les symboles de fonctions, mais au prix d'un effort de démonstration important. Par ailleurs, elles ne s'appliquent qu'à des types isomorphes. Cela nous a conduit au développement de nouvelles tactiques (section 4) qui pourront être utilisées dans d'autres cadres que SMTCoq. L'approche par classes de types proposée dans `ppsimpl` et les *transfer tactics* est néanmoins très intéressante pour l'extensibilité et nous souhaitons la porter à notre cadre.

6 Conclusion et perspectives

La combinaison des deux méthodes présentées dans cet article permet d'étendre SMTCoq avec des tactiques de plus en plus expressives, sans remettre en cause la simplicité et l'efficacité du noyau : il n'a pas besoin de gérer des variables liées ou de multiples représentations des données. Nous pensons que cette approche permet de programmer des tactiques automatiques performantes (en efficacité et en expressivité) avec une interface de haut niveau facilement extensible. Cela fait de SMTCoq un combinateur de tactiques.

En complément de l'implantation de tactiques automatiques, certaines idées présentées dans cet article sont à notre connaissance originales et utilisables dans d'autres contextes. Le fait d'utiliser simultanément des encodages profonds avec des encodages superficiels dans les certificats (§ 3) permet de combiner, dans un même vérificateur certifié, la manipulation de termes par leur structure ou par leur représentation en Coq. Le fait d'appliquer une fonction d'injection aux termes d'un type donné dans tout contexte, grâce à l'introduction d'une variable fraîche (§ 4), permet d'avoir une tactique opérant sous les symboles de fonctions non interprétés.

Les perspectives propres à chaque fonctionnalité ont été présentées tout au long de l'article. De nombreuses extensions peuvent être mises en place pour continuer à améliorer les tactiques automatiques de SMTCoq : l'intégration de nouvelles théories comme les types de données algébriques, les chaînes de caractères ou les ensembles finis ; ou encore l'encodage des aspects d'ordre supérieur. Un de nos objectifs à plus long terme est d'offrir une interface de très haut niveau pour ajouter à Coq de nouvelles procédures de décision dédiées supportées par SMT, se combinant aux procédures de décision existantes de manière agnostique, en se basant par exemple sur les idées de [6].

Remerciements Les auteurs remercient Guillaume Melquiond pour ses conseils sur la définition des tactiques de conversion de la section 4, Claude Marché pour la relecture de cet article, ainsi que les relecteurs pour leurs remarques.

Références

[1] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Certified Programs and Proofs*, pages 135–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[3] Frédéric Besson. ppsimpl : a reflexive Coq tactic for canonising goals. In *CoqPL*, 2017.

[4] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A Learning-Based Fact Selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3) :219–244, 2016.

[5] S. Böhme and T. Weber. Fast LCF-Style Proof Reconstruction for Z3. In *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[6] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Proofs in conflict-driven theory combination. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 186–200. ACM, 2018.

[7] Sylvain Boulmé. Defensive Certification in Coq with ML Type-Safe Oracles. http://www-verimag.imag.fr/~boulme/dc201604, 2016.

[8] Sylvain Boulmé and Alexandre Maréchal. Toward Certification for Free ! working paper or preprint, July 2017.

[9] Lukasz Czajka and Cezary Kaliszyk. Hammer for Coq : Automation for Dependent Type Theory. *J. Autom. Reasoning*, 61(1-4) :423–453, 2018.

[10] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds, and Cesare Tinelli. Extending SMTCoq, a Certified Checker for SMT (Extended Abstract). In *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016.*, pages 21–29, 2016.

[11] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. SMTCoq : A plug-in for integrating SMT solvers into Coq. In *Computer Aided Verification - 29th International Conference*, Heidelberg, Germany, July 2017.

[12] J. Hurd. System Description : The Metis Proof Tactic. *Empirically Successful Automated Reasoning in Higher-Order Logic (ESHOL)*, pages 103–104, 2005.

[13] Chantal Keller. *Proof Technology in Mathematics Research and Teaching*, chapter SMTCoq : Mixing automatic and interactive proof technologies. Springer, 2018. À paraître.

[14] Stéphane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. PhD thesis, University of Paris-Sud, Orsay, France, 2011.

[15] Pierre Letouzey. The zify tactic. https://coq.inria.fr/library/Coq.omega.PreOmega.html.

[16] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[17] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[18] Théo Zimmermann and Hugo Herbelin. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. *CoRR*, abs/1505.05028, 2015.